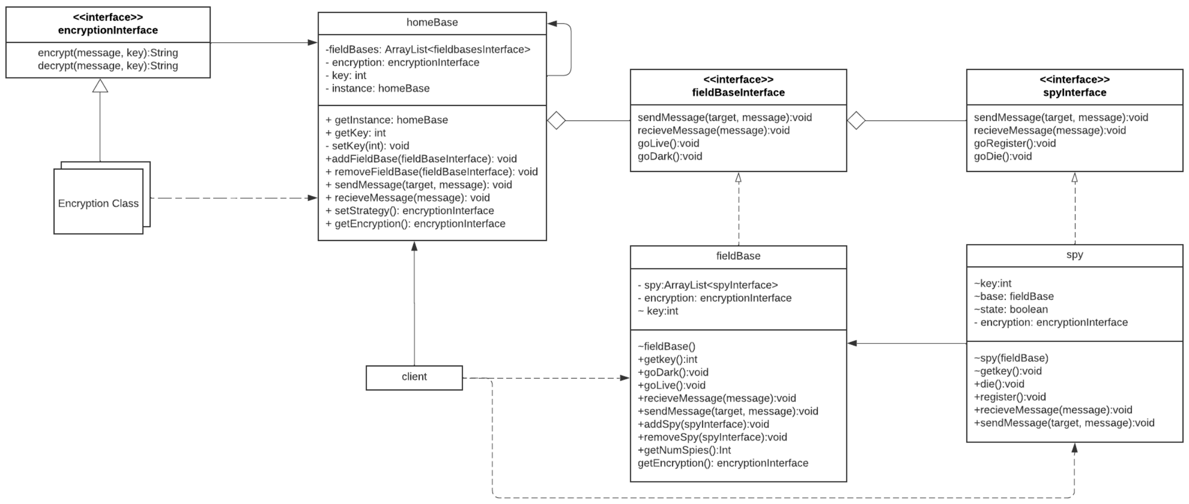
Assignment 2 2ME3

Aryaan Sheth

Nov 17, 2022

Part I

UML Diagram of Design


<<interface>>
encryptionInterface
encrypt(message, key):String
decrypt(message, key):String
Encryption Class
homeBase
-fieldBases: ArrayList<fieldbasesInterface>
- encryption: encryptionInterface
- key: int
- instance: homeBase
+ getInstance: homeBase
+ getKey: int
- setKey(int): void
+addFieldBase(fieldBaseInterface): void
+ removeFieldBase(fieldBaseInterface): void
+ sendMessage(target, message): void
+ recieveMessage(message): void
+ setStrategy(): encryptionInterface
+ getEncryption(): encryptionInterface
client
<<interface>>
fieldBaseInterface
sendMessage(target, message):void
recieveMessage(message):void
goLive():void
goDark():void
<<interface>>
spyInterface
sendMessage(target, message):void
recieveMessage(message):void
goRegister():void
goDie():void
fieldBase
- spy:ArrayList<spyInterface>
- encryption: encryptionInterface
~ key:int
~fieldBase()
+getkey():int
+goDark():void
+goLive():void
+recieveMessage(message):void
+sendMessage(target, message):void
+addSpy(spyInterface):void
+removeSpy(spyInterface):void
+getNumSpies():Int
getEncryption(): encryptionInterface
spy
~key:int
~base: fieldBase
~state: boolean
- encryption: encryptionInterface
~spy(fieldBase)
~getkey():void
+die():void
+register():void
+recieveMessage(message):void
+sendMessage(target, message):void

Design Patterns

Encryption

For the multiple encryption schemes that our design needs to support, I used the **Strategy** pattern as it allowed me to easily swap in and out different behaviors for the encryption schemes. It also allowed to change encryption schemes on the go using setters in the Home Base.

Home Base

For the home base, I chose to use the **singleton** design pattern. This is because the instructions document emphasized the importance of having only one home base. The singleton pattern does exactly that and limits the number of instances of the class to one. This is done by checking the existence of the class in its constructor and only instantiating a new instance of that class the class was initialized to null.

Field Base & Spy

For both these classes, I chose to use a **observer** design pattern. The main reason was for the messaging feature and wanting all other classes to be able to send messages to each other. The observer pattern allows for this to happen. Both the field base and spy classes are built from an interface which allows multiple instance of the class to be created easily.

Future Thoughts

If we wanted to implement a system to ship out multiple home bases around to different locations, we could use the **factory** design pattern. This would allow us to create multiple instances of the home base that come pre-packaged with field bases and spies. To achieve this we would create a wrapper class of the current structure and apply the factory pattern on that.

Design Principals

DRY

My design follows the DRY principle as I use interfaces, abstract classes, and generics to reduce the amount of code that needs to be re-written in the code. For example, the bitshift and decryption methods are implemented in the abstract class and all the classes can call on that which reduces the amount of code that needs to be written within the individual classes.

Open-Close Principal

My design follows the open close principal as it's very modular and it is super easy to add new features and functions to classes via the use of the interfaces and abstract classes that encompass the structure. For example, if we wanted to add a new type of spy that can do something different, we can simply create a new spy that extends the spy class and implement the new methods that are needed.

Design For Interfaces

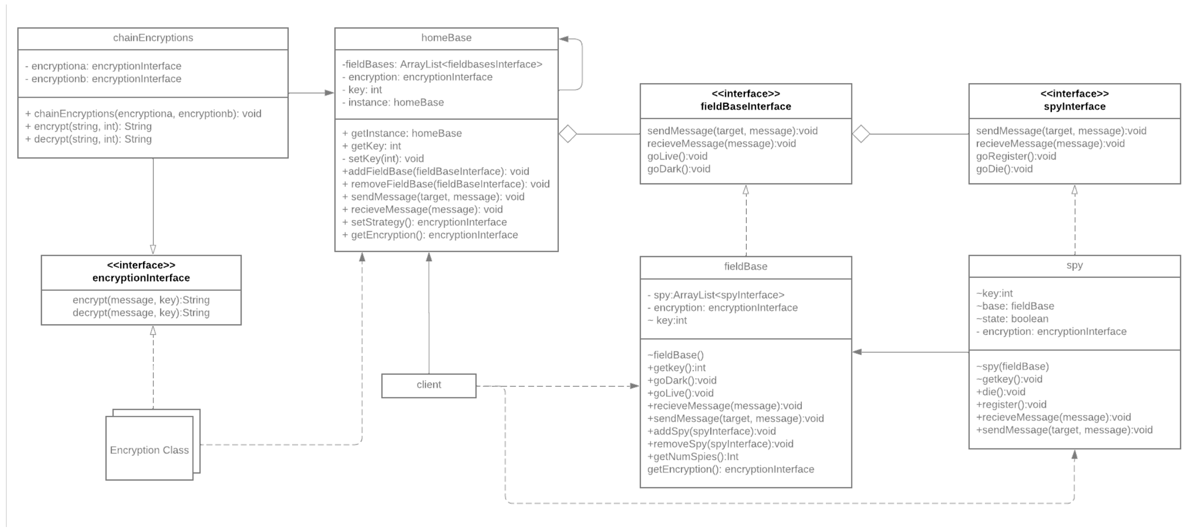
My design follows the design for interfaces principal as all the classes are built from interfaces and their structure is dictated by the interface. For example, the spies and field bases are built from the spy and field base interfaces which dictate the structure of the classes such as their goLive and goDark methods.

Design Intention

My design intention is to create a modular design that allows for easy expansion on existing classes via child classes or using a design pattern such as factory to create multiple instances of the home base design. This is done by using interfaces to create a structure that can be easily expanded on. With the use of the observer pattern, we can easily add new classes and chain them together to create a new subsystem from field bases and agent and expand the overall structure. This also enables the messaging system to work as it allows for the classes to be able to send messages to each other and the home base and be notified of them.

Part II

UML Diagram of New Design


chainEncryptions
- encryptiona: encryptionInterface
- encryptionb: encryptionInterface
+ chainEncryptions(encryptiona, encryptionb): void
+ encrypt(string, int): String
+ decrypt(string, int): String
homeBase
-fieldBases: ArrayList<fieldbasesInterface>
- encryption: encryptionInterface
- key: int
- instance: homeBase
+ getInstance: homeBase
+ getKey: int
- setKey(int): void
+addFieldBase(fieldBaseInterface): void
+ removeFieldBase(fieldBaseInterface): void
+ sendMessage(target, message): void
+ recieveMessage(message): void
+ setStrategy(): encryptionInterface
+ getEncryption(): encryptionInterface
<<interface>>
fieldBaseInterface
sendMessage(target, message):void
recieveMessage(message):void
goLive():void
goDark():void
<<interface>>
spyInterface
sendMessage(target, message):void
recieveMessage(message):void
goRegister():void
goDie():void
<<interface>>
encryptionInterface
encrypt(message, key):String
decrypt(message, key):String
fieldBase
- spy:ArrayList<spyInterface>
- encryption: encryptionInterface
~ key:int
~fieldBase()
+getkey():int
+goDark():void
+goLive():void
+recieveMessage(message):void
+sendMessage(target, message):void
+addSpy(spyInterface):void
+removeSpy(spyInterface):void
+getNumSpies():Int
getEncryption(): encryptionInterface
spy
~key:int
~base: fieldBase
~state: boolean
- encryption: encryptionInterface
~spy(fieldBase)
~getkey():void
+die():void
+register():void
+recieveMessage(message):void
+sendMessage(target, message):void
client
Encryption Class

Modification Of Program

With the updated UML, a new chain encryptions class exists between the encryption interface and the homeBase. This class allows for the homeBase to combine multiple encryption schemes together and apply them to the message. The constructor defines two encryption schemes that are used to encrypt the message. The encrypt method then applies both of these encryption schemes by implementing the encryption interface and overriding the encrypt and decrypt methods. This allows the chaining of encryption schemes to be done easily and without changing any previous code which ensures the open-close principal isn't violated.